AFRL-RI-RS-TR-2013-226

# LIBRARY CODE SECURITY ANALYSIS

*ASPECT SECURITY*

*NOVEMBER 2013*

FINAL TECHNICAL REPORT

**AIR FORCE RESEARCH LABORATORY**
**INFORMATION DIRECTORATE**

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

## NOTICE AND SIGNATURE PAGE

AFRL-RI-RS-TR-2013-226 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

**/ S /**

FRANK H. BORN
Work Unit Manager

**/ S /**

WARREN H. DEBANY, JR.
Technical Advisor, Information
Exploitation & Operations Division
Information Directorate

# REPORT DOCUMENTATION PAGE

*Form Approved*
**OMB No. 0704-0188**

| | | |
|---|---|---|
| **1. REPORT DATE** *(DD-MM-YYYY)* NOV 2013 | **2. REPORT TYPE** FINAL TECHNICAL REPORT | **3. DATES COVERED** *(From - To)* APR 2013 – JUL 2013 |

| | |
|---|---|
| **4. TITLE AND SUBTITLE** LIBRARY CODE SECURITY ANALYSIS | **5a. CONTRACT NUMBER** FA8750-13-C-0126 |
| | **5b. GRANT NUMBER** N/A |
| | **5c. PROGRAM ELEMENT NUMBER** 62788F |
| **6. AUTHOR(S)** Bojan Simic, Arshan Dabirsiaghi, Jeff Williams | **5d. PROJECT NUMBER** INTR |
| | **5e. TASK NUMBER** 00 |
| | **5f. WORK UNIT NUMBER** 02 |
| **7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)** Aspect Security<br>9175 Guilford Road, Suite 300<br>Columbia, MD 21046 | **8. PERFORMING ORGANIZATION REPORT NUMBER** N/A |
| **9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)** Air Force Research Laboratory/RIGA<br>525 Brooks Road<br>Rome NY 13441-4505 | **10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/RI |
| | **11. SPONSORING/MONITORING AGENCY REPORT NUMBER** AFRL-RI-RS-TR-2013-226 |

**12. DISTRIBUTION AVAILABILITY STATEMENT**
Approved for Public Release; Distribution Unlimited. PA# 88ABW-2013-4774
Date Cleared: 14 NOV 2013

**13. SUPPLEMENTARY NOTES**

**14. ABSTRACT**
Eighty percent of code in modern web applications comes from various third party libraries and frameworks and 26% of the most commonly used libraries contain vulnerabilities. According to data gathered analyzing 29.8 million libraries, the majority of library flaws are yet to be discovered and most organizations do not seem to have a process in place for validating or analyzing the open source and third party libraries they use every day. This effort focused on creating a tool that leverages an Interactive Application Security Testing (IAST) tool, Contrast, to identify previously unknown vulnerabilities in Java libraries. This technology will give previously unavailable insight into the security posture of open source libraries that many organizations often falsely assume are secure.

**15. SUBJECT TERMS**
Vulnerabilities, Malware, Application Security, Static Analysis

| **16. SECURITY CLASSIFICATION OF:** | | | **17. LIMITATION OF ABSTRACT** | **18. NUMBER OF PAGES** | **19a. NAME OF RESPONSIBLE PERSON** FRANK H. BORN |
|---|---|---|---|---|---|
| **a. REPORT** U | **b. ABSTRACT** U | **c. THIS PAGE** U | UU | 22 | **19b. TELEPONE NUMBER** (*Include area code)* N/A |

## TABLE OF CONTENTS

## LIST OF FIGURES

## LIST OF TABLES

# 1. EXECUTIVE SUMMARY

The use of software libraries and components in critical applications is skyrocketing, and there has been very little attention to their security. Working with Air Force Research Laboratory, researchers from Aspect Security have created a new approach to analyzing third party Java libraries for vulnerabilities and potential hazards. This report documents the challenges in this endeavor, the tasks performed to create such a tool, and the results from analyzing thirty of the most common libraries downloaded from the Central repository.

Eighty percent of code in modern web applications comes from various third party libraries and frameworks [1]. In 2012, Aspect studied downloads of open source libraries from the Central repository and found that 26% of those downloads were of libraries containing known vulnerabilities [1]. When a library is vulnerable, it opens an application to an attack and could lead to an exploit leveraging the full privilege of the application, sensitive data access, denial of service, as well as executing transactions without authorization. According to data gathered in this earlier study, analyzing 29.8 million libraries, the majority of library flaws are yet to be discovered [1] and most organizations do not seem to have a process in place for validating or analyzing the open source and third party libraries they use every day.

The primary motivation for this research is to take a closer look at the security of open source libraries and frameworks. Where our last research focused on “known” vulnerabilities in libraries, this research targets previously “unknown” vulnerabilities. We hope to gain the ability to automatically determine latent vulnerabilities and hazards that are currently hidden in these libraries. Creating such a tool to would provide greater transparency regarding these libraries’ security posture and allow developers to make better-informed decisions when implementing a library in their software development efforts.

Our approach to building this tool was to combine the idea of “fuzzing,” sending random data to an interface of some sort, with an instrumentation-based vulnerability detector. For the fuzzing piece, we created a custom fuzzing framework that overcomes some of the challenges with speed and scale associated with fuzzing such a large Application Programming Interface (API) and complex data structures. For the detection engine, we leveraged the work Aspect has performed previously building Contrast [4]. Aspect has successfully used Contrast to find vulnerabilities in web applications and this seemed to be a logical application of the technology. This approach allowed us to do “deep” analysis of methods, including not only the code of the method itself, but also all the code invoked within the scope of that method.

We applied this tool to the latest version of 31 of the most popular Java libraries, including web frameworks and security libraries. These libraries are extremely widely used, and have presumably received the most security scrutiny of any of the open source libraries. The results of the experiment were encouraging. These libraries comprised over 110,000 methods. In them, we found a total of 19 vulnerabilities in 4 different libraries. These flaws range from weak encryption algorithms to path traversal. In addition, we identified over 2,300 hazards – these are not full vulnerabilities but possibly dangerous effects of methods that developers should be aware of when using a library.

While we did not achieve the code coverage that we had hoped for, we believe that this technique is promising. We hope that this work can eventually lead to a market for third-party components that makes it possible to take security into account when building critical systems.

# 2. INTRODUCTION

Organizations have long been taking steps to improve the code quality and overall security posture of their custom written code. However, very seldom do these organizations pay significant attention to the security of the innumerable libraries being imported into their Java projects every day. Companies often falsely assume that the open source libraries they are using contain secure and high quality code due to their widespread adoption in the software development industry. In addition, the library code used in an application does not generally get updated when a new version of that library becomes available. If the new version contains patched vulnerabilities then the applications running the previous version will contain further known vulnerabilities.

Libraries often have full access to each layer of a software product including business functions, data access, and resource management. The average Java application utilizes more than 30 libraries that typically comprise 80% of the application code. A major concern is that although a project may be utilizing a significant number of libraries, only a small portion of each one is actually utilized by the programmer. Given this fact, the important question is how do we know the rest of the library is not doing something malicious? Since library use is so widespread, there exists a need to have the ability to analyze and test the functionality. Doing this analysis on an entire library allows us to check it for vulnerabilities and potential hazards such as denial of service.

## 2.1 Vulnerabilities and Hazards

This study makes a distinction between "vulnerabilities" and "hazards." We consider a method vulnerable if a developer is likely to call that method with data that could cause an undesirable outcome. For example, if a developer calls the java.sql.Statement.execute() method with data that includes input from the user, this code is considered vulnerable to a Structured Query Language (SQL) injection attack.

A "hazard," on the other hand, is a method that has an unexpected outcome that may or may not cause an undesirable outcome. For example, the output from the Throwable.getMessage() call may often include some of the user's input. Although it is generally unknown to developers and is not always dangerous, it is a potential hazard. Developers should know that the message from this call is "tainted" with user input, and should be handled carefully. In this study, we focused only on whether a method was a propagator of tainted data. There are many other hazards that it might be possible to explore with this technique.

## 2.2 Related Works

There exist few previous efforts in the area of library security analysis. This is largely because most information security efforts are in the area of identifying and mitigating vulnerabilities in custom code. The most relevant research done on library security was the paper "Unfortunate Reality of Insecure Libraries" by Aspect Security. This research brings to light the magnitude at which organizations use insecure libraries every day. Aspect Security analyzed 31 of the most frequently downloaded libraries across multiple versions and established that 26% of them had known vulnerabilities in their latest version. This paper was a motivating factor for our research because it sparked an interest in what other unknown vulnerabilities may exist.

There have been several efforts to do fuzzing of Java programs. One particularly interesting paper was by Karthick Jayaraman and David Harvison in their effort to create a white box fuzzer for Java called JFuzz [5]. The JFuzz tool is a concolic fuzzer built on top of the NASA Java PathFinder project.

JFuzz uses a combination of concrete and symbolic execution to do constraint solving in its effort to do whitebox fuzzing. We believe that this tool is very powerful and useful for fuzzing smaller libraries since the constraint solving portion can be very time consuming. In the future it may be beneficial to use JFuzz to do the constraint-solving portion of the analysis separately. Once the constraint analysis is completed, we can use those results as input to achieve better code coverage.

Another interesting paper on whitebox fuzzing called “Grammar-Based Whitebox Fuzzing” written by researchers at Microsoft and Massachusetts Institute of Technology (MIT) contains great ideas on how to achieve greater code coverage [6]. The writers take a grammar-based approach to overcome many of the problems with whitebox testing such as complex structured inputs. This grammar-based approach allows for much better coverage of control paths and was evaluated using with the Internet Explorer Version 7 JavaScript interpreter program. We believe that such an approach for Java based programs could benefit our own research efforts by increasing code coverage.

## 2.3 Interactive Application Security Testing

The Contrast Interactive Application Security Testing (IAST) tool developed by Aspect Security is targeted towards not only identifying vulnerabilities in custom code but also in libraries. Reference [4] explains how this security technology works. The application is able to track data throughout its lifecycle inside an application and is able to analyze whether that data pathway could be used for malicious intent. Data can be tracked as various imported libraries use and execute it. Contrast then looks for vulnerabilities such as injection, improper encryption algorithms, and weak data validation implementations to notify and inform an organization. We decided to use this software in our effort to locate vulnerabilities because of its comprehensive rules engine and ability to provide meaningful results without the static of false positives.

## 3. METHODS, ASSUMPTIONS, AND PROCEDURES

Analyzing Java libraries for security vulnerabilities and hazards required several tasks. The first was to create a methodology for isolating all the classes and methods inside the library that needed to be tested. The second task was to generate a fuzzer that would exercise the desired classes and methods. The fuzzer's function is to provide the largest possible code coverage of each library analyzed. The third task was to add the Contrast security testing software to the fuzzing process in order to detect vulnerabilities and hazards. These findings are based on the set of rules that allow Contrast to locate vulnerabilities at execution time. The fourth and final task was to exercise thirty-one of the most commonly used Java libraries and to analyze the data gathered for vulnerabilities and hazards.

### 3.1 Tasks

To address task one, exercisable classes and methods were restricted to public or protected methods inside each class. Interfaces and abstract classes were also included in the testing whenever possible. The exercise of these classes was accomplished by searching through the classes loaded in the JVM that implemented or extended these interfaces or abstract classes. Once a match was found, the program attempted to instantiate the implementing class in order to exercise its methods. In order to test all of these classes, all dependencies needed to be loaded into the class loader. These dependencies were downloaded from the central Maven repository programmatically using the Eclipse Aether tool for working with repositories [10]. Once the program is provided the name of the JAR file to be exercised, all necessary dependencies are downloaded from Maven and loaded into the classpath.

For task two, a robust fuzzer was implemented in order to get the most code coverage possible. Because there are practically unlimited inputs to exercise the library with, we established a maximum time period of 10 minutes for fuzzing. The fuzzer attempts to create an instance of each class chosen to be exercised by using its various or methods that return an instance of that particular class. Once an instance is created, it is stored in an object pool and re-used as necessary. This process was critical to achieving higher code coverage since data loaded into the objects located in the object pool are persisted throughout the analysis of the entire library. The resulting code coverage showed an average of 42% of methods, 36% instructions, and 18% of lines being exercised throughout our testing of thirty-one Java libraries.

Adding Contrast to the analysis and exercise process of the libraries allowed us to leverage the taint propagation and security rules defined in the tool. Each parameter passed to a method that was to be exercised was marked as tainted. In the event that the method returned a value that was tainted, our program marked that method as potentially hazardous. A method would return a tainted value if no validation or encoding was completed on the data according to Contrast's rules. This also provided us with detection capabilities for vulnerabilities such as SQL Injection, weak cryptography, path traversal, and APIs vulnerable to denial-of-service (DOS) attacks.

To the fourth task, thirty of the most commonly downloaded Java libraries were exercised. The results are first stored into a database and then exported to Comma Separated Values (CSV) format. The database data included information on each method exercised: its corresponding class information, any hazards or vulnerabilities detected, and code coverage statistics based on lines, methods, and instructions. As part of the Contrast output, the Extensible Markup Language (XML) data for each hazard and vulnerability detected was also included in the results. This included the stack trace leading to each of the vulnerabilities made available for triage and validation purposes.

## 3.2 Architecture of Fuzzing Tool

The library analysis tool created in this effort is comprised of several key components. This section will outline the role of each component created to deliver vulnerabilities and hazards contained inside libraries. It will also describe how each component works and the technology utilized to create it.

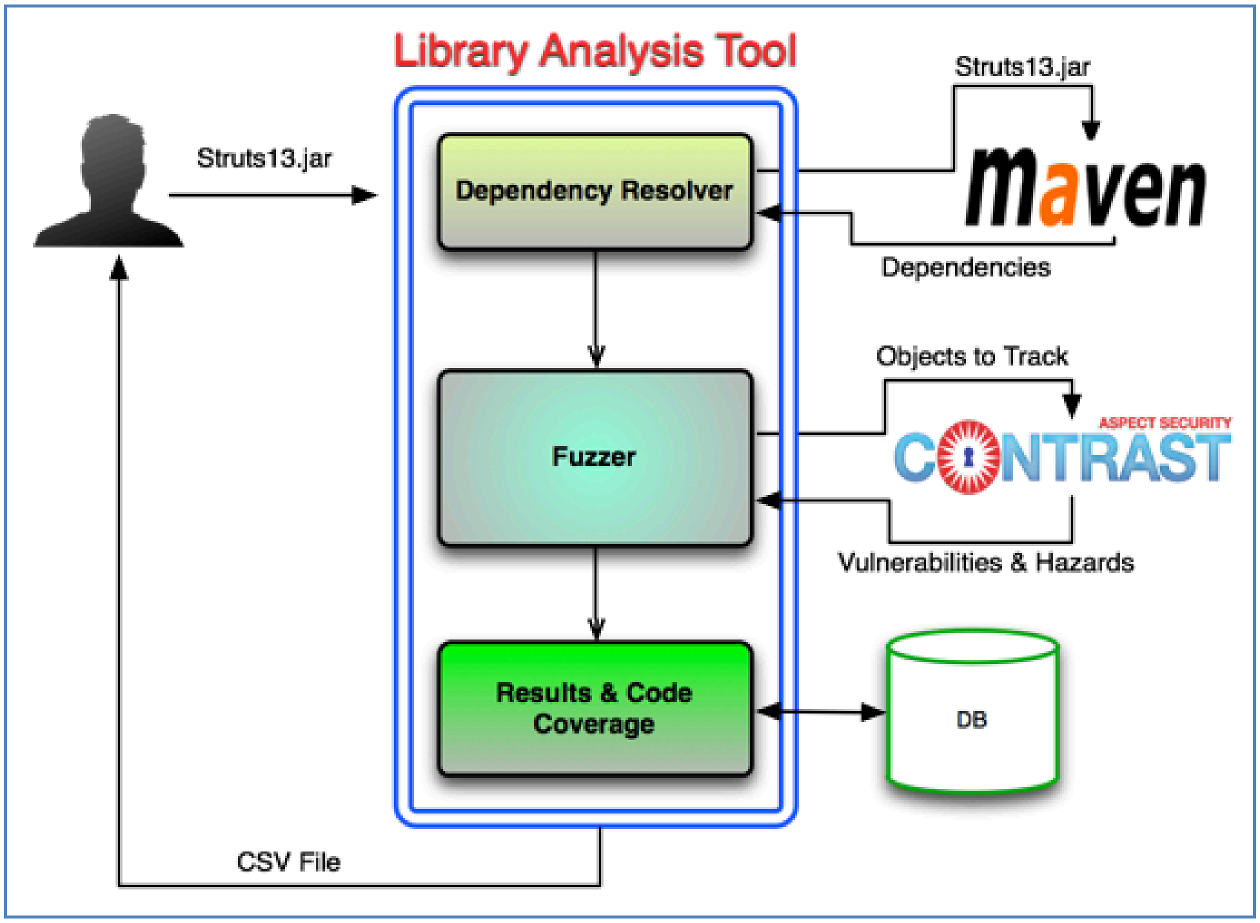


Figure 1 - Process for Library Analysis

## 3.3 Method for Dependency Resolution

In order to exercise a library's functionality, all dependencies for that library must be resolved and loaded onto the classpath of the Java Virtual Machine. We have chosen to use a tool called Aether [10] to programmatically retrieve all dependencies for the library to be analyzed. Each dependency for the library was downloaded into a newly created Maven repository and then loaded onto the classpath at runtime. This approach ensures that all dependencies are resolved when the library's classes and methods are being exercised. In the event that a library analyzed at a later time uses an already downloaded dependency, the necessary file was re-used.

## 3.4 Method for Fuzzing

Once a library's dependencies are resolved, a fuzzer is necessary to get the most code coverage. The implemented fuzzer attempts to create instances of each class inside a library that contains public or protected methods. Constructing these instances is a recursive process since the creation of an instance typically requires instances of various objects specified as parameters.

The simplest iteration in creating the fuzzer was exercising classes and methods that only take primitive types as arguments. For this task we chose to leverage a library called DummyCreator [3] that creates instances of primitive types. For more complex fuzzing that requires instances of complex objects, a custom solution was implemented. The fuzzer consists of several components called "Makers" that are responsible for creating an instance of a class by either using its constructor, methods, or class type. In the event that a class is abstract or an interface, the corresponding Maker object will attempt to locate implementing classes and instantiate them instead. This methodology allows us to get much larger code coverage because we are also attempting to instantiate abstract classes and interfaces that are commonly passed as argument types.

For each method being fuzzed, multiple instances of its parameters are created and then permutated to generate a random set of arguments. Once the various permutations are generated, each method is executed a thousand times if there are enough permutations. The execution of these methods is spread through fifteen separate threads that are assigned tasks to execute a particular method. This number was chosen via trial-and-error due to performance considerations and memory consumption observed throughout testing. It was determined that executing methods for all possible permutations resulted in out of memory errors. The decision to limit method execution to one thousand also demonstrated that code coverage was not affected by a significant percentage.

## 3.5 Method for Code Coverage

Every time a class is to be exercised, it is loaded into a component called a "Coverage Tracker". This component utilizes the Java Code Coverage Library (JaCoCo) [8] library for determining code coverage. Once the class is loaded into the coverage tracker, each line of code that is exercised from its methods is tracked. The coverage tracker also tracks information such as total methods executed, instructions executed, and branches executed. For the purpose of this research, we are mainly interested in the number of methods and lines of code executed in our calculation of code coverage. Once the exercise of a class is completed, its code coverage data is stored and added to the overall code coverage for that particular library.

## 3.6 Method for Contrast Integration

The Contrast tool has the capability to locate vulnerabilities inside Java Web Applications. This is typically done by tracking data throughout execution and analyzing the code using this data. To integrate Contrast with our tool, each class, method, and parameter needed to be tracked. Prior to the exercise of a class' methods, an instance of that method's declaring class and all its arguments were provided to the Contrast engine. The engine then tracked the data throughout the remainder of that library's analysis.

Whenever a method is exercised, all the data that is passed to that method is marked as tainted. If that method returns a value, it must be checked to determine whether or not it is still tainted. In the event that the return value contains tainted data, we mark this method as a potential hazard because we cannot be sure that the returned value isn't malicious. Contrast determines if the data is tainted by applying a complex set of rules to check for events such as validation or escaping.

Contrast also provides vulnerability analysis for each method exercised. For example, if a method makes a call to System.exit() or uses an ineffective encryption algorithm, Contrast will create an output displaying the trace of the vulnerability inside the library. The vulnerability analysis provided by the Contrast engine is “deep,” meaning that not only the code of the single method being fuzzed is examined, but all the code within the scope of that method, including other libraries and the Java runtime itself.

## 3.7 Method for Result Storage and Export

The results gathered from exercising each library are stored in three separate database tables. The “JAR Coverage” table contains information about code coverage for each library exercised. The JAR Coverage data includes:

- Total lines of code inside the library
- Lines of code successfully executed
- Total methods inside the library’s classes
- Number of methods successfully executed
- Total instructions inside the library’s code
- Number of instructions successfully executed
- Total branches inside the library’s code
- Number of successful branches explored

The second database table contains method execution data and is called the “Method Tracker”. This table contains:

- Name of each method exercised
- The declaring class of each method
- Total number of execution attempts for that method
- Number of times that method was successfully executed
- Number of Hazards that method resulted in

The third and final database table is called “Hazards” and it contains all the hazards and vulnerabilities determined by Contrast. The table contains the following data:

- Method name that resulted in the hazard
- Corresponding class name
- Rule ID that shows the rule that is associated with that hazard or vulnerability
- Name of the library that the hazard or vulnerability belongs to
- The hash value assigned by Contrast
- The XML trace of the hazard that displays the exact location of the hazard

Once all results are stored in the database, there is functionality to export the database tables into CSV format that can then be imported into a processing tool such as Microsoft Excel.

## 3.8 Technologies

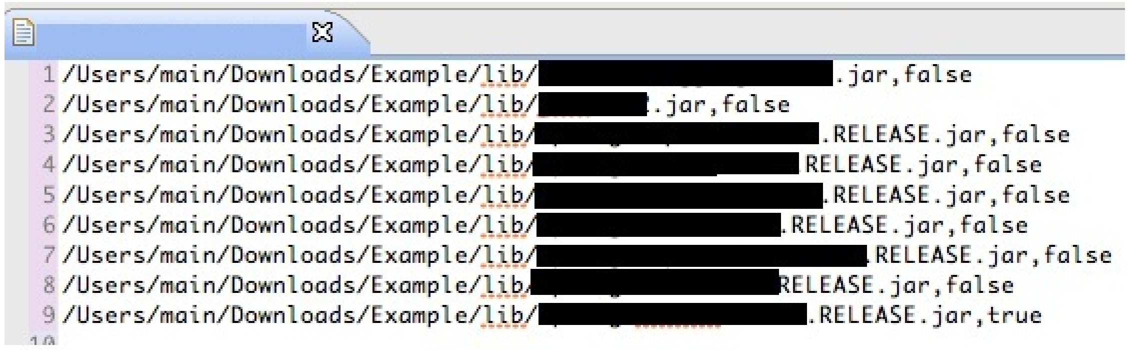
Since our effort is to analyze Java libraries for security vulnerabilities, we chose to implement our tool with version 1.6 the Java programming language. Some of the other tools in the effort included:

- Eclipse Juno – Development environment
- Maven – Dependency management
- Aether – Managing dependencies programmatically.
- Mockito [7] – A mocking framework for Java
- Java Reflections Library [2] – Dynamic Method invocation
- Contrast Engine – Vulnerability and hazard detection
- Java DummyCreator library [3] – Creating mock objects for primitive types
- JaCoCo – Tracking code coverage
- ObjectDB – Database for storing result data
- JUnit – Unit testing of components
- Log4j – Logging
- Subversion – Version control system

## 3.9 Implementation of User Interface

For this project we implemented a simple command line user interface. The user simply needs to provide a few pieces of information regarding the library they want to analyze and the program handles the rest. This input string consists of the Group ID, Artifact ID, and version of the library to be analyzed from Maven's central repository. An example of this sample input string looks like "org.apache.struts:struts-core:jar:1.3.10". The third portion of the string specifies that we want to download the "JAR" file for that particular library.

The command line interface also allows the user to provide a second argument. This argument is a path to an input file that has the location of the library the user wants to analyze. This file should also contain the paths to all dependencies that the library may depend on. A Boolean flag at the end of each entry in the file tells the program whether or not it is a dependency or if it is to be analyzed. A user may utilize this feature if they want to analyze a custom library that does not exist in the Maven repository. This feature can also be used in the event that there is no network connection and Maven cannot be used to resolve dependencies.



```
/Users/main/Downloads/Example/lib/[redacted].jar,false
/Users/main/Downloads/Example/lib/[redacted].jar,false
/Users/main/Downloads/Example/lib/[redacted].RELEASE.jar,false
/Users/main/Downloads/Example/lib/[redacted]RELEASE.jar,false
/Users/main/Downloads/Example/lib/[redacted].RELEASE.jar,false
/Users/main/Downloads/Example/lib/[redacted].RELEASE.jar,false
/Users/main/Downloads/Example/lib/[redacted].RELEASE.jar,false
/Users/main/Downloads/Example/lib/[redacted]RELEASE.jar,false
/Users/main/Downloads/Example/lib/[redacted].RELEASE.jar,true
```

Figure 2 - Example input file

Once the user starts the program and provides the correct input, the program will attempt to download the library as well as all its dependencies. In the event of a failure, the program will simply stop execution. Failures can occur if the input string is not valid, the library does not exist, or there is no network connection that can resolve the library information with Maven. If the user provided an input file, the program will validate that all the files exist and will continue execution. When using the input file approach, the program will assume that all dependencies are valid and that no other dependencies are necessary.

Throughout the program's execution the user will be given information on which method is currently being exercised. This information consists of how many successful executions have occurred as well as any failures encountered. Once all possible methods are executed, the program will provide three separate CSV files as output. These files will be prefixed by the name of the library and will contain information on hazards found, method execution details, as well as an overall summary with code coverage data. These files will be created in the same directory as the library analysis tool.

The library analysis tool will also generate a file called report.odb. This file contains the same data as the three CSV files but can be opened using the ObjectDB [9] explorer tool. Using this tool, the user can write SQL queries to explore the result data if desired.

## 3.10 Extracting Methods and Classes for Fuzzing

Once the program reconciles the input library and all its dependencies, we had to determine exactly what methods to exercise and how to accomplish this successfully. The first task was to load the library we want to analyze onto the classpath along with all its dependencies. Instead of writing a custom solution for doing runtime metadata analysis of libraries, we chose to use an open source solution called Reflections [2].

The Reflections library contains functionality to get information such as methods, fields, constructors, and other metadata about a JAR file. This library can take as a parameter a number of "Scanners" that act as filters when searching for particular metadata within a JAR file. It was soon determined that these scanners were not robust enough to retrieve all the data that was needed for our purposes. The source code of the Reflections library was downloaded and several modifications were made. One such modification was to add the ability to extract all methods for every class in an entire JAR file. With the modifications, the Reflections library has the ability to take a JAR file as input and return a list of objects that contain every method as well as what its declaring class is.

The modified version of the Reflections library was then re-compiled and imported into the library analysis project. After extracting all the methods and classes from the JAR the user wishes to analyze, we were able to choose only public and protected methods. The project would then use this list to exercise all the methods inside it.

## 3.11 Implementing the Fuzzer

The initial iterations of the library analysis project consisted of only fuzzing methods that had primitive types as parameters. The initial fuzzer would create multiple instances of primitive types and then store them into an object pool for later use. These primitive types contained normal and boundary values. For example, when creating instances of an integer, it would be assigned Integer.MAX_VALUE, Integer.MIN_VALUE, 0, 1, and -1 value. Using this approach resulted in low code coverage and it was determined that a fuzzer for more than just primitive types was necessary.

In order to increase code coverage, we implemented a fuzzer that can create complex objects. Initially, a tool called DummyCreator [3] was used to create both primitive and complex types. However, it was soon determined that when working with complex types, the tool often times failed and was extremely unreliable. It was then decided to use the DummyCreator tool to only create instances of primitive types since the values were more random. For complex types, a custom solution was necessary.

This custom solution for fuzzing methods that take complex types consisted of the creation of objects called “Makers”. A maker was created for constructors, methods, abstract types, interfaces, and arrays. These various Makers would attempt to create an instance of a class by using its various methods or constructors. If the class is abstract, the corresponding maker would attempt to locate a subclass and recursively try to instantiate that object. In the event of an interface, the maker would attempt to locate an implementing class somewhere on the classpath and try to instantiate that object. Another tool that is used at a limited capacity is Mockito [7]. This tool provides the ability to create mock objects for objects that make use of interfaces or abstract classes in the event that our other methods are exhausted.

When an instance of a class is created, it is put into an object pool so that it can be used whenever it is needed. The fuzzer tries to create multiple instances of each class in order to achieve better code coverage during code exercise. Since all instances of objects were stored in a pool, the same objects were often re-used. This re-use allowed for the modifications to the tainted object to be persisted throughout the fuzzing process. When the fuzzer attempts to exercise a method, it retrieves the necessary parameters from the object pool. It then creates all possible permutations of these objects and executes the method with instances from the permutation pool.

One behavior that was observed when exercising methods with random permutations of its parameters is a slight inconsistency in the results. This inconsistency is small enough to be negligible but can be useful if the user wants to analyze a library over several iterations using randomized values.

The fuzzer executes each method a maximum of 1000 times. The exact number of executions depends on the amount of permutations created. The number of permutations depends on the amount of the method’s arguments. For example, if the method has zero parameters, it will only be executed once. But if it has 10 parameters, it will be executed 1000 times since executing 10! Permutations would cause various memory and performance issues. The limit of 1000 executions was determined after analysis of the program’s memory consumption. Since the method execution is done in a separate thread, a maximum timeout of 30 seconds is enforced in order to prevent deadlock. If the thread takes more than 30 seconds to exercise a method, it is interrupted and given another task. The need for this limit was determined after extremely long execution periods for certain methods (greater than 2 hours).

## 3.12 Integration with Contrast Engine

The fuzzer had to be integrated with the Contrast tool's engine component in order to retrieve hazard and vulnerability data. The Contrast engine component is able to track a piece of data throughout its lifecycle in an application. For this project, prior to exercising a method, we tell Contrast to track all the method's parameters. If a method returns a value, we ask Contrast if the value returned is tainted. If so, we mark that particular method as a hazard.

Contrast has a set of complex rules that it applies to every piece of code that is executed. These rules check for the use of unsafe method calls and libraries. If Contrast finds the execution of unsafe code, it reports a finding. The finding contains some very useful information such as the method, the stack trace for the unsafe execution, and the Contrast rule that was broken.

## 3.13 Code Coverage and Results

Code coverage was implemented using the Java Code Coverage Library (JaCoCo). This library is able to track the entire contents of a Java class and determine exactly what code was executed down to the line number level. Prior to using JaCoCo, code coverage was only calculated at the method level by manually calculating the total methods in a class and then determining which were successfully executed.

Results from Contrast consisted of an XML string that contained the trace of any hazard/vulnerability reported. The library analysis tool uses an object database (ObjectDB) to store all of the code coverage and result data. This database is then queried and the results are written to three separate CSV files.

## 4. RESULTS AND DISCUSSION

Our analysis consisted of 31 separate JAR files that contained some of the most popular web application frameworks, security libraries, and most frequently downloaded utilities. This section will discuss in detail the behavior observed throughout the analysis process and discuss the accuracy of the results. The table below will outline the overview of data collected throughout our analysis.

Table 1 - Overall data metrics captured by analysis

| | |
|---|---|
| Number of methods executed | 47,279 |
| Lines of code executed | 60,090 |
| Number of instructions executed | 920,998 |
| Branches of code explored | 8,038 |
| Total successful method executions | 252,612 |
| Total attempted method executions | 457,953 |
| Vulnerabilities reported by Contrast | 18 |
| Hazards reported by Contrast | 2325 |

### 4.1 Vulnerabilities Reported

The Contrast tool that was incorporated into the library analyzer reported 19 potential vulnerabilities inside of the 31 JAR files exercised. These findings were located in:

Table 2 - Vulnerabilities identified

| JAR File | Vulnerability | Number of Instances |
|---|---|---|
| Undisclosed JAR #8 | • Banned API<br>• Unsafe Readline | 1<br>1 |
| Undisclosed JAR #25 | • Bad Use of Media Access Control (MAC)Cryptography<br>• Weak random number generation | 13<br>2 |
| Undisclosed JAR #17 | • Directory Path Traversal | 1 |
| Undisclosed JAR #29 | • XML eXternal Entity (XXE) | 1 |

The above table shows that potential vulnerabilities were reported in only 12.9% of the libraries analyzed. This observation is definitely positive since it is less than the 26% average discovered in Aspect's research last year with regard to Common Vulnerabilities and Exposures (CVE's). However, it is important to mention that the libraries studied are the most scrutinized in existence, and only the latest versions of these libraries were analyzed. Older versions would more than likely contain a higher percentage of vulnerabilities.

## 4.2 Hazards Reported

Throughout our analysis, libraries were exercised in an out of context manner. This means that a library such as Struts, was exercised simply by invoking its public and protected methods only. The library was not part of a web application with a functioning server or valid configuration files typically provided to a framework. The role of the “Hazard” in our results is to notify the user whether or not they can trust the data received from a method in each library analyzed. When a class receives some tainted data and the methods in that class are exercised, a hazard will be reported if the return value of the method is tainted. Taint is determined by Contrast’s ability to follow the data throughout its lifecycle and checking whether or not it has been escaped or validated in a proper way.

Contrast has the ability to track a tainted piece of data until it is garbage collected. If a piece of data is split or merged into another object, the resulting data is marked as tainted and tracked separately. For each hazard reported, we are able to see a trace of the data in order to better understand exactly how the library may be using any provided input.

Analysis of 31 libraries resulted in 2325 potential hazards.

Table 3 - Number of Hazards for Analyzed Libraries

| JAR File Name | Number of Hazards |
|---|---:|
| Undisclosed JAR #1 | 4 |
| Undisclosed JAR #2 | 11 |
| Undisclosed JAR #3 | 408 |
| Undisclosed JAR #4 | 57 |
| Undisclosed JAR #5 | 58 |
| Undisclosed JAR #6 | 32 |
| Undisclosed JAR #7 | 2 |
| Undisclosed JAR #8 | 52 |
| Undisclosed JAR #9 | 3 |
| Undisclosed JAR #10 | 12 |
| Undisclosed JAR #11 | 287 |
| Undisclosed JAR #12 | 722 |
| Undisclosed JAR #13 | 88 |
| Undisclosed JAR #14 | 43 |
| Undisclosed JAR #15 | 55 |
| Undisclosed JAR #16 | 5 |
| Undisclosed JAR #17 | 38 |
| Undisclosed JAR #18 | 39 |
| Undisclosed JAR #19 | 47 |
| Undisclosed JAR #20 | 15 |
| Undisclosed JAR #21 | 53 |
| Undisclosed JAR #22 | 11 |
| Undisclosed JAR #23 | 1 |
| Undisclosed JAR #24 | 11 |
| Undisclosed JAR #25 | 24 |
| Undisclosed JAR #26 | 82 |
| Undisclosed JAR #27 | 14 |

| | |
|---|---|
| Undisclosed JAR #28 | 8 |
| Undisclosed JAR #29 | 22 |
| Undisclosed JAR #30 | 19 |
| Undisclosed JAR #31 | 102 |
| **Total** | **2325** |

## 4.3 Code Coverage

After analyzing 31 JAR files, the average code coverage with respect to method executions is 42%. However, code coverage based off of line executions, is only 18.15%. There are several factors that could attribute to this behavior. One assumption is that the code could contain significant amounts of boilerplate code or the general bloating of Java classes. It is also important to remember that the total method count shown in table 2 below includes private methods that were not part of our analysis. Line number values include non-executable code such as attribute definitions and import statements.

Table 4 - Code coverage by Jar File

| Jar File Name | Successful Methods | Total Methods | Method Percentage | Successful Lines | Total Lines | Lines Percentage |
|---|---|---|---|---|---|---|
| Undisclosed JAR #1 | 25 | 203 | 12.32% | 33 | 803 | 4.11% |
| Undisclosed JAR #2 | 431 | 1161 | 37.12% | 509 | 1794 | 28.37% |
| Undisclosed JAR #3 | 15366 | 27105 | 56.69% | 17733 | 91994 | 19.28% |
| Undisclosed JAR #4 | 15030 | 27531 | 54.59% | 16990 | 92155 | 18.44% |
| Undisclosed JAR #5 | 253 | 1373 | 18.43% | 633 | 5555 | 11.40% |
| Undisclosed JAR #6 | 1649 | 3673 | 44.90% | 3607 | 12820 | 28.14% |
| Undisclosed JAR #7 | 161 | 1234 | 13.05% | 384 | 4114 | 9.33% |
| Undisclosed JAR #8 | 3 | 236 | 1.27% | 14 | 618 | 2.27% |
| Undisclosed JAR #9 | 29 | 108 | 26.85% | 71 | 276 | 25.72% |
| Undisclosed JAR #10 | 494 | 2158 | 22.89% | 1439 | 10800 | 13.32% |
| Undisclosed JAR #11 | 1190 | 3985 | 29.86% | 400 | 1659 | 24.11% |
| Undisclosed JAR #12 | 4486 | 11606 | 38.65% | 1029 | 3497 | 29.43% |
| Undisclosed JAR #13 | 989 | 2177 | 45.43% | 2135 | 9716 | 21.97% |
| Undisclosed JAR #14 | 600 | 1892 | 31.71% | 1789 | 11240 | 15.92% |
| Undisclosed JAR #15 | 203 | 571 | 35.55% | 601 | 2308 | 26.04% |
| Undisclosed JAR #16 | 306 | 1012 | 30.24% | 836 | 4804 | 17.40% |
| Undisclosed JAR #17 | 376 | 2090 | 17.99% | 1110 | 11987 | 9.26% |
| Undisclosed JAR #18 | 653 | 2382 | 27.41% | 1653 | 9218 | 17.93% |
| Undisclosed JAR #19 | 536 | 1450 | 36.97% | 1338 | 7387 | 18.11% |
| Undisclosed JAR #20 | 1043 | 2934 | 35.55% | 2005 | 10734 | 18.68% |
| Undisclosed JAR #21 | 517 | 3815 | 13.55% | 1004 | 13743 | 7.31% |
| Undisclosed JAR #22 | 13 | 87 | 14.94% | 32 | 550 | 5.82% |
| Undisclosed JAR #23 | 1 | 2 | 50.00% | 1 | 8 | 12.50% |
| Undisclosed JAR #24 | 88 | 220 | 40.00% | 13 | 310 | 4.19% |
| Undisclosed JAR #25 | 178 | 388 | 45.88% | 324 | 1664 | 19.47% |
| Undisclosed JAR #26 | 456 | 2245 | 20.31% | 964 | 8213 | 11.74% |

|  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|
| Undisclosed JAR #27 | 400 | 611 | 65.47% | 1314 | 3088 | 42.55% |
| Undisclosed JAR #28 | 20 | 136 | 14.71% | 87 | 683 | 12.74% |
| Undisclosed JAR #29 | 318 | 1062 | 29.94% | 966 | 5394 | 17.91% |
| Undisclosed JAR #30 | 403 | 1049 | 38.42% | 1076 | 3935 | 27.34% |
| Undisclosed JAR #31 | 1062 | 5881 | 18.06% | 0 | 0 | 0.00% |
| Totals | 47279 | 110377 | 42.83% | 60090 | 331067 | 18.15% |

After analyzing the code coverage for each Jar file, we determined that web application frameworks such as Stripes, Java Server Faces (JSF), and Struts showed some of the higher code coverage percentages. For some of the libraries such as Xerces, the JaCoCo tool used to measure code coverage failed to track the amount of lines in the library. This fact could potentially skew the averages to some degree.

The lack of higher code coverage can be attributed to several factors:

- The fuzzer is unable to instantiate contextual objects. Many frameworks and libraries depend on contextual objects that contain session information and contextual data. Since we are unable to create these objects programmatically to fit the context of the individual library, code coverage suffers.
- The fuzzer does not have access to various properties and input files necessary for many libraries to operate. Several libraries such as Struts or Spring depend on XML input files to provide critical mapping information. The fuzzer does not have access to these files and is unable to generate them at runtime.
- The fuzzer cannot instantiate many web specific objects. When instantiating objects such as HttpRequest, often times the randomly generated data passed to the constructors is incorrect and results in an invalid object.
- Lack of memory and computing power. Since there are potentially millions of permutations generated of each set of arguments passed to a constructor or method. Executing each one of these permutations became impossible due to constant out of memory issues. Therefore it was decided to only run a maximum of 1000 randomly chosen permutations per method.

Code coverage data was also tracked in terms of instructions and branches. The average percentage of instructions covered was 36.72 and only 6.30 in terms of branches.

## 5. CONCLUSIONS

A previous study done by Aspect Security researchers has determined that 26% of library downloads have known vulnerabilities. Static and dynamic analysis tools lack the capability to exercise the majority of code in libraries referenced by an application even though this is often 80% of the overall source. Using a custom developed fuzzer along with Contrast, we have created a program that analyzes the security of Java libraries. This program is able to exercise a significant portion of a library's source code.

The development of this tool required the use of several technologies for fuzzing, measuring code coverage, as well as memory and performance management. The resulting program provides the user with valuable hazard and vulnerability data that may have previously been unknown. We have analyzed thirty Java libraries and uncovered 2325 potential hazards and 19 possible vulnerabilities. This program can be used to help an organization determine the security posture of the libraries they use every day.

We anticipate that there is much left to do in the effort to ensure the use of secure libraries. Our program is able to provide users with a previously unattainable visibility into a library's security posture. With increasing focus on library security, we hope to continue our research to give programmers the ability to make educated decisions when developing software.

### 5.1 Opportunities

We believe encouraging more secure libraries and components in the software marketplace is one of the most critical cyber security challenges there is.  Although in our research significant progress was made, there is much more work to be done in the effort to analyze Java libraries for security. The first task would be to collect more data by analyzing other popular libraries. It would also be extremely important to analyze previous versions of libraries. The data collected by analyzing older versions of libraries would tell organizations whether or not their legacy applications are at risk.

In order to provide more useful data and better code coverage of libraries, it would be necessary to improve our fuzzing technology. To improve coverage, the fuzzer would need to provide contextual information that libraries require. This includes various property and configuration files as well as mocking of objects that have valid data. If we are able to provide libraries with valid objects that are necessary for proper execution, it would ensure greater code coverage and provide us with the ability to discover more vulnerabilities.

Performance and memory management is another area in need of improvement. Code coverage would improve with the ability to execute methods with more permutations of mocked objects. Currently we limit a method's execution to 1000 times with permutations of only a few objects due to performance reasons. This task could be accomplished improving memory management as well as distributing the programs execution over multiple CPU's.

For more accurate fuzzing, the creation of a Genetic Algorithm (GA) could ensure much greater code coverage. The GA would learn which objects are successful in instantiating a class or executing a method. This improvement in accuracy with the fuzzer would greatly increase both performance and code coverage. Valuable memory space would be saved by garbage collecting all unnecessarily created instances in the object pool and freeing up space for other more useful data. Performance would also increase due to the GA being able to learn when a method has been fully executed. Learning when a method's branches and instructions have been explored would allow us to limit the executions count of these methods and free up threads to run queued tasks.

## 6. REFERENCES

[1] Aspect Security - The Unfortunate Reality of Insecure Libraries

https://www.aspectsecurity.com/uploads/downloads/2012/03/Aspect-Security-The-Unfortunate-Reality-of-Insecure-Libraries.pdf

[2] Reflections – Java Runtime Metadata Analysis

http://code.google.com/p/reflections

[3] DummyCreator - A Java-Library to Create Instances of Any Class

http://code.google.com/p/dummycreator

[4] Contrast – Security for JavaEE That Just Works

https://www.aspectsecurity.com/wp-content/plugins/download-monitor/download.php?id=154

[5] JFuzz – A Concolif Whitebox Fuzzer for Java

http://people.csail.mit.edu/vganesh/Publications_files/vg-NFM2009-jFuzz.pdf

[6] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In PLDI, pages 206–215, 2008.

[7] Mockito – A Mocking Framework for Java

http://code.google.com/p/mockito

[8] JaCoCo code coverage library – Java Code Coverage Library

http://www.eclemma.org/jacoco/trunk

[9] ObjectDB – Object Database for Java

http://www.objectdb.com/java/jpa/tool/explorer

[10] Aether – Repository API for Maven

http://www.sonatype.org/aether

## 7. ACRONYMS

| | |
|---|---|
| API | Application Programming Interface |
| CSV | Comma Separated Values |
| CVE | Common Vulnerabilities and Exposures |
| DB | Database |
| DOS | Denial of Service attack |
| GA | Genetic Algorithm |
| IAST | Interactive Application Security Testing |
| IE7 | Internet Explorer 7 |
| JaCoCo | Java Code Coverage Library |
| JAR | Java Archive |
| JSF | Java Server Faces |
| MAC | Media Access Control |
| MIT | Massachusetts Institute of Technology |
| SQL | Structured Query Language |
| XML | Extensible Markup Language |
| XML | Extensible Markup Language |
| XXE | XML External Entity |